An Effective Access Control Scheme for Preventing Permission Leak in Android

Longfei Wu[1], Xiaojiang Du[1], and Hongli Zhang[2]
[1]Department of Computer and Information Science, Temple University, Philadelphia, PA, USA, 19122
[2]School of Computer Science and Technology, Harbin Institute of Technology, Harbin, China, 150001
Email: [1]{longfei.wu, dux}@temple.edu, [2]zhanghongli@hit.edu.cn

Abstract—**In the Android system, each application runs in its own sandbox, and the permission mechanism is used to enforce access control to the system APIs and applications. However, permission leak could happen when an application without certain permission illegally gain access to protected resources through other privileged applications. We propose SPAC, a component-level system permission based access control scheme that can help developers better secure the public components of their applications. In the SPAC scheme, obscure custom permissions are replaced by explicit system permissions. We extend current permission checking mechanism so that multiple permissions are supported on component level. SPAC has been implemented on a Nexus 4 smartphone, and our evaluation demonstrates its effectiveness in mitigating permission leak vulnerabilities.**

Keywords—Permission leak; access control; smartphone security

I. Introduction

Android is a privilege-separated operating system built upon a customized linux kernel. The Android system assigns each application a unique user ID (Uid) for identification purpose, and applications are running in separated processes as a way of isolating them from each other and from the system. By default, applications cannot interact with each other and have limited access to the system resources.

Android enforces permission-based mechanisms to provide a fine-grained access control to system resources and third-party applications. Specifically, sensitive system APIs are protected by system permissions, and third-party applications can make use of these APIs by first requesting the corresponding permissions in its manifest file. At the beginning of installation process, all requested permissions are presented to the user. If the user agrees to complete the installation, all those requested permissions are to be granted. Applications may also define and enforce their own permissions, which is called custom permissions. All custom permissions can specify one of the four protection levels: *normal, dangerous, signature, signatureOrSystem* (detailed in Section II-B). The custom permissions, as well as system permissions, can be used to protect third-party applications. An application can specify a certain permission that client applications must have for interaction, by setting the *android:permission* attribute of the *application* element (for all components) or of a particular component in the manifest file. It is also possible for an application to check caller's permissions during runtime, which is embedded in its source code.

Components are the essential building blocks of an Android application. Each component is encapsulated as its own entity, and performs a specific function. There are four different types of components: *activity*, *service*, *content provider* and *broadcast receiver*. Components can communicate to each other (inter-components communication) through an message passing mechanism called *Intent*. A component can be specified as either public or private by setting the *exported* property in the manifest file. Private components can only be accessed by components within the same application, while public components are accessible by other applications. Public components can be protected either statically, by declaring permision requirement in the manifest file, or dynamically, by performing permission checking during runtime.

In this paper, we first present the weakness of custom permissions regarding permission leak attacks. Then we propose system permission based access control (SPAC) scheme, which provides straightforward and fine-grained permission enforcement on component level. SPAC can mitigate permission leak vulnerabilities by utilizing system permissions instead of obscure custom permissions, which establishes accurate one-to-one matching between each permission and corresponding sensitive system API. The custom permission is adopted in SPAC only when its protection level is *signature* for restriction of access to applications signed with the same signature. Besides, SPAC can also detect potential permission leaks among applications sharing the same user ID. Such collaborative permission augmentation is prevented by enforcing the required permissions on each of the associated applications. Compared to policy-based access control schemes, SPAC is lightweight as it only extends the permission checking module to enable multiple permissions to be enforced. We have implemented SPAC on a Nexus 4 smartphone running Android 4.2 OS, and conducted experiments to evaluate its performance. Our results show that SPAC is very effective in protecting APIs against permission leak.

The rest of the paper is organized as follows. Section II introduces the motivation of our work. Section III presents the design of SPAC scheme. Section IV shows the implementation of SPAC and the proposed permission checking algorithms. Section V presents the performance evaluation. Section VI discusses related works. Section VII concludes the paper.

II. Motivation

A. Permission Leak

Permission leak vulnerability has been studied in many previous works. The particular attack specified to exploit per-

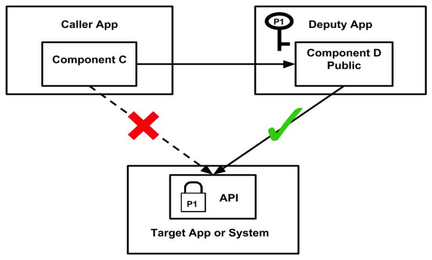


Fig. 1. The Permission Leak Model

mission leak vulnerability can be called a permission escalation attack [1] or permission re-delegation attack [2]. The motive of these attacks is obvious: an application with no/few sensitive permissions is unlikely to be suspected. The attack model can be stated as:

A non-privileged caller application without required permissions illegally gains access to protected resources.

The attack model is described in Figure 1. As it shows, the application programming interface (API) protected by permission *P1* could either belong to the Target App or the system. The Caller App is not granted with *P1* so it cannot access the API directly by itself. However, the Deputy App owns the permission *P1*, and the target API is accessible by all its components. In this scenario, if component D of the Deputy App is public and not guarded by any permission, then the Caller App is able to call the target API indirectly through that component D even though without permission *P1*.

The Caller App can successfully launch the permission escalation attack since the Deputy App delegates its request, intentionally or inadvertently. In this paper, we focus on mitigating permission leaks in inter-components communication. Specifically, we are concerned about helping Android developers better protect the public components, so that permission leak is prevented in the inter-components communication with caller applications.

B. Custom Permission

Misconfigured deputy is a major cause of permission leak problem. However, the definition of “misconfiguration” is quite obscure in previous works. In [2], they parse the manifest file to identify at-risk applications before performing IPC inspection. And their policy takes all public components protected by permissions as safe. Similarly, components in the absence of declarative or dynamic permission checking that can be invoked by other applications (with different *Uid*) are regarded as misconfigured in [3]. DroidChecker [4] targets at applications use at least one permission while contain unprotected components that are publicly visible. Zhou et al. [5] study the passive content leaks and pollution in *content provider*. The candidate applications they select include those guarded by custom permissions at *normal* level since the system will automatically grant permissions at this level without user’s explicit approval. But in fact, the unreliability of custom-defined permissions is only partially revealed.

There are four types of protection levels. Since we focus on third-party deputy applications, the custom permissions we deal with are at *normal*, *dangerous* and *signature* protection level. Apart from *normal* level, we continue digging into the other two protection levels, and identify the weakness of custom permissions.

A *dangerous* level permission is considered safer as it needs user’s explicit approval if requested. During installation, the custom permissions are presented to the user all along with other system permissions requested. However, the user may not well understand the name and description of custom permissions since they are not explicitly mapped to system services or device resources as system permissions do. Also the user could ignore the custom permissions even if he/she is confused, since all permissions are required to be granted to install an application.

A *signature* level permission is believed to completely eliminate the possibility of permission leak beccause only applications signed with the same certificate can conduct inter-components communication. A signature is owned by each developer and will not leak to others. However, collusion attacks could happen among applications packaged by the same attacker (using the same signature). Therefore, although permission at *signature* protection level can be used to prevent the access of other developers’ applications, it cannot guarantee the absence of permission or data leakage.

Besides, custom permissions can obscure the capability of an application. System permissions are well-regulated: detailed information on the meaning and functionality of each permission is publicly available. By contrast, custom permissions cannot be matched correspondingly to system actions or resources, and may be used to cover the actual capability and intention of malicious applications. Hence, due to the huge gap between custom permissions and system permissions, applications using custom permissions are still under the risk of permission leak.

III. The SPAC Scheme

A. Our Design

We propose *SPAC*, a system permission based access control scheme to mitigate permission leak vulnerabilities in Android inter-components communication. For each public component, developers are asked to replace the custom-defined permissions in their applications (if any) with corresponding system permissions. In the transition, the corresponding system permissions include, on one hand, the permissions required to access protected system APIs by this particular component; on the other hand, if it can communicate with other intra-application components, permissions needed by other components are also included. Generally, we can group the (commonly requested) protected system APIs into three categories: privileged actions, sensitive data and system notifications (broadcasts). Figure 2 gives an illustration of the permission transition in SPAC scheme.

In Figure 2, Deputy App has three components: private Component 1, public Component 2 and private Component 3. The five system APIs are protected by permission P1, P2, P3, P4 and P5, respectively. Component 1 can invoke the privileged operation Action 1; Component 2 is able to receive system Broadcast 1, access to sensitive Data 1 and invoke Action 2; Component 3 can access to sensitive Data 2. Originally, no permission is enforced on Component 1 and 3 since they are private and not accessible by other applications, while public

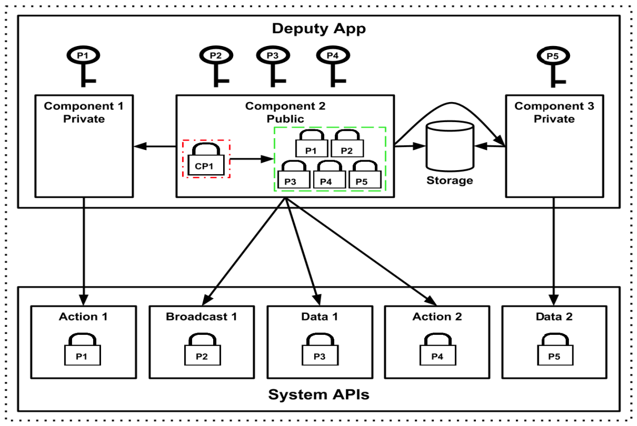


Fig. 2. Illustration of the Permission Transition in SPAC Scheme

Component 2 is solely protected by a custom permission CP1. Note that Component 2 can also access the intra-application components Component 1 and 3.

In the conversion to system permissions set ($SPset$), the corresponding permissions of system APIs accessible directly by Component 2 are first added (P2, P3 and P4). Then, the required permissions of other reachable APIs via intra-application components are also added (P1 and P5). Hence, as described in Figure 2, the custom permission CP1 is replaced by a combination of system permissions in SPset.

$$SPset = Permissions_{dir} \bigcup Permissions_{indir}$$

The only exception is when the custom permission is at *signature* protection level. In this special case, access to the component is intended to be restricted to application signed by the same developer. We need to keep this custom permission in the final permission set ($FPSet$) to retain this feature after the permission transition.

$$FPset = SPset \bigcup Custom\_Permission_{sig}$$

The API call may be invoked in a call-chain manner which contains more than just one deputy application. As shown in Figure 3, Component C in Caller App invokes along the whole call path (Deputy App 1 to n) to reach a system API. Following the rule of permission transition, each intermediate Deputy App between the system API and Calling App has to enforce the permission protecting the system API on their public accessible components. The invocations in Figure 3 are all direct invocations, but there may exist routes to access other system APIs via intra-application components. So we can have

$$Permission_{API} \subseteq FPset_{D_n} \subseteq ... \subseteq FPset_{D_1} \subseteq FPset_C$$

A broadcasted *Intent* may be used for instant notification, or may contain private information that the sender does not want to be intercepted by unexpected broadcast receivers. In Android system, a broadcast can enforce permission requirement as well. The permission check must be passed for the *Intent* to be delivered to target receivers. Similarly, a broadcast may leak permission to receivers if it is associated with sensitive events or data, but is unproperly protected by custom permission. Hence, SPAC scheme is also applied to broadcasts.

IV. Implementation

In SPAC scheme, the public accessible components of third-party applications that are unprotected or solely protected

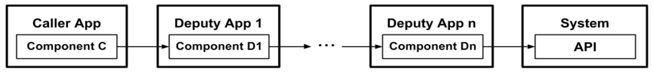


Fig. 3. The Call Chain Model

by unreliable custom permissions need to perform permission transition, so that system permissions can effectively mitigate attacks targeted at permission leak. For each of such components, the corresponding system permissions set is composed of all system APIs it can access directly and via other components in the same application. So it is highly possible that multiple system permissions are enforced on the component. A special case is when the protection level of the original custom permission is *signature*, both custom permission and system permissions are required to protect the component collaboratively. The *android:permission* attribute of *application* element or a component can be set to the permission that clients must have in order to interact with the associated object. However, the Android system allows only one such attribute for the application as well as each component.

The current Android permission mechanism has to be extended so that it can support multiple required permissions to be specified for each component, or the whole application (for all components). In addition, existing broadcast *Intent* also can only handle one optional required permission that a receiver must have. The intuitive solution is to modify the parsing policy of the manifest file as well as the internal structure of each type of components, so that more than one associated permission requirement can be recognized and enforced. This method also needs to expand the parameter structure of system functions to process broadcasts with multiple required permissions. Obviously, this method is too heavy since the data structure in all relevant functions are to be changed.

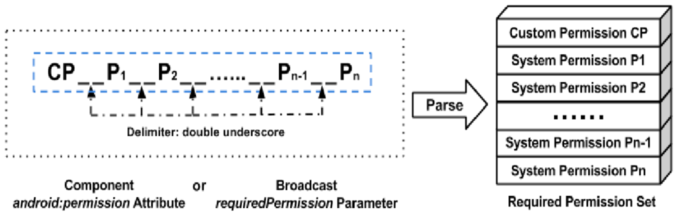


Fig. 4. Permission Parsing in SPAC Scheme

We choose another "clever" way to enable multiple permission enforcement on components and broadcasts. We want to keep using the current data structures for permission enforcement, which include the single attribute *android:permission* in component elements and the single *requiredPermission* string in functions dealing with broadcasts. Instead of creating more attributes or parameters, all permissions to be enforced can be specified in the existing single attribute or parameter. A new format is defined in which permissions can be concatenated and later recovered through parsing. The permission attributes in manifest file only accept letters, numbers, period and underscore as valid character, and all of them are commonly used in permission names for identification purpose (e.g. "android.permission.READ_SMS"). So we select double underscores as the delimiter to seperate consecutive permissions. All the permissions stay combined together until the permission checking is performed, where the permission being checked is first parsed into segments of single system

Algorithm 1 *Uid*PC: *Uid*-level Permission Checking

```
Input:  Combined Permission $P$, Target User ID $Uid$;
Output: Result $r$;
1: Split the combined permission $P$ into an array of single permissions $P_{sgl}[]$ with double underscores as delimiter.
2: Initialize result $r$ as GRANTED.
3: for each permission $P_{sgl}[i]$ do
4:     Check the presence of $P_{sgl}[i]$ in the target $Uid$;
5:     if $P_{sgl}[i]$ is absent then
6:         Set result $r$ to DENIED;
7: Return $r$;
```

and custom permissions as shown in Figure 4. Then all these permissions are checked individually for a given *Uid*. The absence of any specific permission will result in the denial of access. The *Uid*-level permission checking algorithm (*Uid*PC) is described in Algorithm 1. Note that it is unnecessary to declare the "combined" permission since it is not intended to take effect in access control. But the seperated *signature* level custom permission (if any) has to be declared.

Meanwhile, since Android is built on a user-based permission model in which permissions are associated with *Uid*, the default permission enforcement mechanism only checks whether a given permission has been assigned to the application's associated *Uid* instead of the application itself. This will make a difference when multiple applications share the same *Uid* by setting *android:sharedUserId* attribute to the same value. The potential threat is that user may be tempted to install malicious (collusive) applications that share the same *Uid*, by which the capabilities of these applications are mutually augmented and become the union set of permissions granted to each one of them. To prevent permission leak among applications with the same *Uid*, we need to enhance the permission checking such that each of these applications should have requested all permissions they need. We further propose application-level permission checking algorithm (AppPC) as in Algorithm 2. The sharing of *Uid* is very common among stock applications, so we first pick out the harmless stock applications that are installed under the "$\backslash$*system*$\backslash$*app*" directory. For the rest of the third-party applications, we check if there are multiple applications associated with the target *Uid*. If not, it is equivalent to *Uid*-level permission checking; Otherwise, we need to ensure every associated application owns all required permissions.

We implement SPAC scheme on a Google Nexus 4 smartphone running Android 4.2 operating system. We modify the source code of Android permission checking mechanism so that it is able to support SPAC with minimal impact to the original Android system.

V. Case Studies and Evaluation

To demonstrate the effectiveness of SPAC scheme, we conduct case studies to evaluate the effectiveness of the two proposed permission checking schemes, respectively.

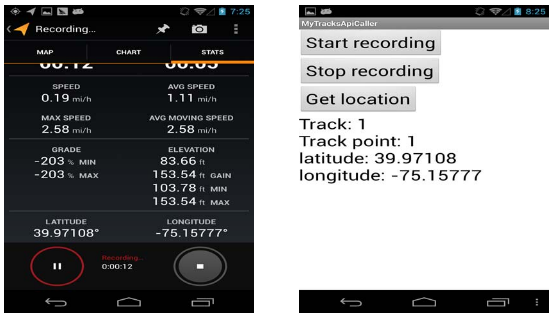
A. Custom Permission

My Tracks [6] is a health and fitness application developed by Google. It can record user's moving path, speed, distance, elevation and location information while doing outdoor

Algorithm 2 AppPC: Application-level Permission Checking

```
Input:  Combined Permission $P$, Target User ID $Uid$;
Output: Result $r$;
1: Obtain the names of all applications that are associated with $Uid$, and store in array $App\_name[]$.
2: if The source directory of the first application $App\_name[0]$ is not in "/system/app/" then
3:     Set $counter$ to 0;
4:     Split the combined permission $P$ into an array of single permissions $P_{sgl}[]$ with double underscores as delimiter.
5:     for each permission $P_{sgl}[i]$ do
6:         for each application $App\_name[i]$ do
7:             Check the presence of $P_{sgl}[i]$ in $App\_name[i]$;
8:             if $P_{sgl}[i]$ is granted then
9:                 $counter$++;
10:    if $counter = App\_name.length \times P_{sgl}.length$ then
11:        Set result $r$ to GRANTED.
12:    else
13:        Set result $r$ to DENIED.
14: Return $r$;
```

sports. Meanwhile, *My Tracks* application provides APIs which allow third-party Android applications to access its data and start/stop a recording. The three public APIs are protected by custom permissions detailed in Table I. We take the Mytracks content provider as an example to illustrate the permission leak in *My Tracks*. We develop an application *MyTracksApiCaller* which only owns the "com.google.android.apps.mytracks.READ_TRACK_DATA" permission to access the content provider. The label and description of this custom permission are both "read Google My Tracks data", which fails to give any direct hint on the potential leakage of location information. But in fact, *MyTracksApiCaller* can successfully obtain the GPS coordinates from *My Tracks* even though it does not have any location-related permissions. Figure 5(a) is the screenshot of *My Tracks* recording panel where we can see the GPS coordinates (39.97108, -75.15777). As shown in Figure 5(b), *MyTracksApiCaller* is able to get the coordinates through the content provider API.



SPEED 0.19 mi/h
AVG SPEED 1.11 mi/h
MAX SPEED 2.58 mi/h
AVG MOVING SPEED 2.58 mi/h
GRADE -203 % MIN -203 % MAX
ELEVATION 83.66 ft 153.54 ft GAIN 103.78 ft MIN 153.54 ft MAX
LATITUDE 39.97108°
LONGITUDE -75.15777°
Recording... 0:00:12

Track: 1
Track point: 1
latitude: 39.97108
longitude: -75.15777

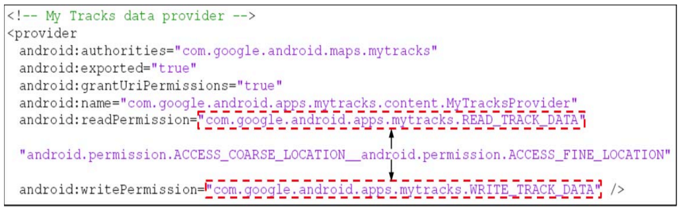
(a) My Tracks (b) MyTracksApiCaller

Fig. 5. Illustration of Permission Leak in *My Tracks*

To mitigate the leak of location permissions, *My Tracks* could follow SPAC scheme and use multiple system permissions to restrict the access of its APIs. In this case, the *dangerous*-level custom permissions are to be replaced by corresponding system permissions shown in Figure 6. After that, unless the caller application has both ACCESS_FINE_LOCATION and ACCESS_COARSE_LOCATION permissions, the access to content provider would be denied.

TABLE I. PROTECTION OF APIS IN *My Tracks*

API	Custom Permission Enforced	Description/Label	Protection Level
MyTracks Content Provider	com.google.android.apps.mytracks.READ_TRACK_DATA com.google.android.apps.mytracks.WRITE_TRACK_DATA	read Google My Tracks data write data to Google My Tracks	*dangerous*
MyTracks Service	com.google.android.apps.mytracks.WRITE_TRACK_DATA	start/stop Google My Tracks recording	*dangerous*
MyTracks Notifications	com.google.android.apps.mytracks.TRACK_NOTIFICATIONS	receive Google My Tracks notifications	*dangerous*



```
<!-- My Tracks data provider -->
<provider
  android:authorities="com.google.android.maps.mytracks"
  android:exported="true"
  android:grantUriPermissions="true"
  android:name="com.google.android.apps.mytracks.content.MyTracksProvider"
  android:readPermission="com.google.android.apps.mytracks.READ_TRACK_DATA"

  "android.permission.ACCESS_COARSE_LOCATION__android.permission.ACCESS_FINE_LOCATION"

  android:writePermission="com.google.android.apps.mytracks.WRITE_TRACK_DATA" />
```

Fig. 6. Permission Transition of *My Tracks*

B. Sharing User ID

To evaluate the effectiveness of the application-level permission checking, we develop two applications App1 and App2 that are signed by the same certificate and declare to share the same *Uid*. App1 has READ_CONTACTS permission and App2 has SEND_SMS permission. Since the two applications run under the same user ID, their granted permissions both appear to be the union of permissions they own individually. As a consequence, App1 can take use of App2's permission to send SMS while App2 can borrow App1's permission to access contacts. However, as shown in Figure 7, application-level permission checking can defend against such kind of permission leak by checking the permissions requested by each of the associated application.

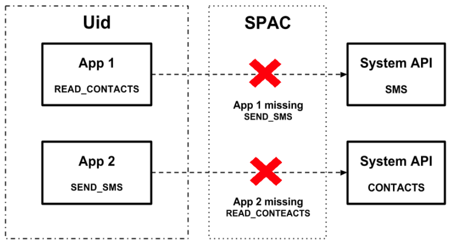


Fig. 7. Illustration of Application-level Permission Checking

VI. RELATED WORK

Permission leak vulnerabilities on Android platform has been studied in previous works. Based on the way to solve the problem, we can categorize them into the following three categories: runtime monitoring [2][7], policy enforcement [8][9], and information flow tracking [4][10][11].

While most previous works adopt various techniques to detect and block permission leak, SPAC mainly focuses on providing more effective and fine-grained access control for application developers to protect their APIs. Saint [8] solves the issue with s similar idea as its policies can be enforced on component level. But Saint constructs a separated policy enforcement mechanism which requires substantial changes to the system including modifiying application installer and parser, placing hooks to the four types of components, embedding additional *AppPolicy Provider* and *FrameworkPolicyManager*, etc. By contrast, SPAC is much more lightweight since it just extends the existing permission checking mechanism. There is no need for extra manifest file, source files or changes to SDK. Besides, we propose to replace the obscure custom permissions with explicit system permissions, while policy based defending approaches are even more confusing as developers need to follow the complicated policies.

VII. CONCLUSION

To mitigate the permission leak vulnerabilities, we propose a component-level system permission based access control (SPAC) scheme, in which explicit and fine-grained system permissions are utilized to protect public APIs of third-party applications. SPAC is lightweight as we extend the existing permission parsing and checking mechanism with minimum modification to the system. We implement SPAC scheme on real device and the performance evaluation proves that it's effective to defend against permission leak attacks.

ACKNOWLEDGEMENT

This research was supported in part by the US National Science Foundation (NSF) under grant 1065444, and by the US Army Research Office under grant WF911NF-14-1-0518.

REFERENCES

[1] L. Davi, A. Dmitrienko, A.-R. Sadeghi, and M. Winandy, “Privilege escalation attacks on android,” in *Proceedings of the 13th International Conference on Information Security*, 2011.

[2] A. P. Felt, H. J. Wang, A. Moshchuk, S. Hanna, and E. Chin, “Permission re-delegation: Attacks and defenses,” in *Proceedings of the 20th USENIX Conference on Security*, 2011.

[3] D. Sbirlea, M. Burke, S. Guarnieri, M. Pistoia, and V. Sarkar, “Automatic detection of inter-application permission leaks in android applications,” *IBM Journal of Research and Development*, Nov 2013.

[4] P. P. Chan, L. C. Hui, and S. M. Yiu, “Droidchecker: Analyzing android applications for capability leak,” in *Proceedings of the Fifth ACM Conference on Security and Privacy in Wireless and Mobile Networks (WISEC)*, 2012.

[5] Y. Zhou and X. Jiang, “Detecting passive content leaks and pollution in android applications,” in *Proceedings of the 20th Network and Distributed System Security Symposium (NDSS 2013)*, Feb 2013.

[6] My Tracks, https://play.google.com/store/apps/details?id=com.google.android.maps.mytracks.

[7] S. Bugiel, L. Davi, A. Dmitrienko, T. Fischer, A.-R. Sadeghi, and B. Shastry, “Towards taming privilege-escalation attacks on android,” in *19th Annual Network & Distributed System Security Symposium (NDSS)*, Feb. 2012.

[8] M. Ongtang, S. McLaughlin, W. Enck, and P. McDaniel, “Semantically rich application-centric security in android,” in *Computer Security Applications Conference (ACSAC)*, Dec 2009.

[9] S. Smalley and R. Craig, “Security enhanced (se) android: Bringing flexible mac to android.” in *20th Annual Network and Distributed System Security Symposium, NDSS*, Feb. 2013.

[10] L. Lu, Z. Li, Z. Wu, W. Lee, and G. Jiang, “Chex: Statically vetting android apps for component hijacking vulnerabilities,” in *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2012.

[11] M. C. Grace, Y. Zhou, Z. Wang, and X. Jiang, “Systematic detection of capability leaks in stock android smartphones,” in *19th Annual Network and Distributed System Security Symposium, NDSS*, Feb. 2012.